Searching

Linear : c*n

Binary: c*$\log_2 n$

Sorting:

Bubble: n^2

Linear: bestcase: c2*n

Worst: c1*n^2

Merge: n*$\log_2 n$

Merge sort (pseudo-code)

```
if data has 0 or 1 element(s) then
    return a copy of data
else
    Sort the first half of data and assign the result to fst;
    Sort the second half of data and assign the result to snd;
    Assign an empty list to res;
    Assign 0 to fi and si;
    while fi and si are valid indexes in fst and snd, respectively do
        if fst[fi] < snd[si] then
            Append fst[fi] to res;
            Increase fi by one;
        else
            Append snd[si] to res;
            Increase si by one;
        end
    end
    if fi is a valid index in fst then
        Glue the rest of fst (from fi onwards) to the end of res ;
    else if si is a valid index in snd then
        Glue the rest of snd (from si onwards) to the end of res ;
    end
    return res
end
```

zipping


```
1  def linear(data, value):
2      """Return the index of 'value' in 'data', or -1 if it does not occur"""
3      # Go through the data list from index 0 upwards
4      i = 0
5      # continue until value found or index outside valid range
6      while i < len(data) and data[i] != value:
7          # increase the index to go to the next data value
8          i = i + 1
9      # test if we have found the value
10     if i == len(data):
11         # no, we went outside valid range; return -1
12         return -1
13     else:
14         # yes, we found the value; return the index
15         return i
```

Algorithm 2: Binary search in an ordered list

input : Ordered, non-empty *data* and target value *val*
output: The index of *val* in *data*, or -1 if *val* does not occur in *data*

```
1  Assign to low and high the first and last index in data;
2  while interval between low and high non-empty and data[low] ≠ val do
3      Choose mid in the middle between low and high ;
4      if data[mid] = val then
5          Assign to low the value of mid ;                          /* Found!  */
6      else if val < data[mid] then
7          Assign to high the value of mid − 1;          /* val lies before index mid  */
8      else
9          Assign to low the value of mid + 1;           /* val lies after index mid  */
10     end
11 end
12 return if data[low] = val then low, otherwise −1
```

Algorithm 3: Ordering a list using *bubble sort*

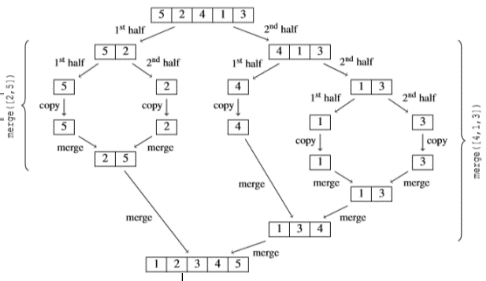
input : List *data*
output: Sorted copy of list *data*

```
1  Assign a copy of data to res ;                          /* res is the result list */
2  Assign the last index in res to ui ;        /* ui is the index of the last unsorted value */
3  while ui > 0 do                                  /* If ui equals 0, everything is sorted */
4  |  Assign 0 to si ;                          /* si is the index of the last swapped value */
5  |  Assign 0 to i ;                          /* i is the index that is currently processed */
6  |  while i < ui do                          /* If ui is reached then the rest is ordered */
7  |  |  if res[i] > res[i+1] then                        /* res[i] en res[i+1] are inverted */
8  |  |  |  Swap res[i] and res[i+1] ;
9  |  |  |  Assign i to si ;                              /* We just swapped at index i  */
10 |  |  end
11 |  |  Increment i by one;
12 |  end
13 |  Assign si to ui ;                              /* The list is ordered from si+1 */
14 end
15 return res
```

3//2 = 1

3/2 = 1.5

3%2 = 1



Algorithm 4: Ordering a list using *merge sort*

input : List *data*
output: Sorted copy of list *data*

```
1  if data has 0 or 1 element(s) then
2  |  return a copy of data
3  else
4  |  Sort the first half of data and assign the result to fst ;
5  |  Sort the second half of data and assign the result to snd ;
6  |  Assign an empty list to res ;                          /* res is the result list */
7  |  Assign 0 to fi and si ;          /* fi and si are increasing indexes in fst and snd  */
8  |  while fi and si are valid indexes in fst and snd, respectively do
9  |  |  if fst[fi] < snd[si] then                          /* fst[fi] is smaller than snd[si] */
10 |  |  |  Append fst[fi] to res ;
11 |  |  |  Increase fi by one;
12 |  |  else                                /* snd[si] is smaller than (or equal to) fst[fi] */
13 |  |  |  Append snd[si] to res ;
14 |  |  |  Increase si by one;
15 |  |  end
16 |  end
17 |  if fi is a valid index in fst then
18 |  |  Glue the rest of fst (from fi onwards) to the end of res ;  /* All snd-elements have been
   |  |  treated */
19 |  else if si is a valid index in snd then
20 |  |  Glue the rest of snd (from si onwards) to the end of res ;  /* All fst-elements have been
   |  |  treated */
21 |  end
22 |  return res
23 end
```

Binary faster when:

For 1000 distinct words

10*N+10000 < 1000*N

990N>10000

N>=10 When searching 10 times or more

Algorithm 5: Removing duplicates

input : Sorted list *data*
output: Copy of *data* in which each value occurs exactly once

```
1  Assign an empty list to res;
   ;                                                        /* res is the result list */
2  if data is not empty then                    /* If data is empty, data[0] does not exist */
3  |  Assign data[0] to fresh ;          /* fresh is a fresh value not yet occurring in res  */
4  |  Assign 1 to i;
5  |  while i is a valid index in data do                  /* Continue until the end of data  */
6  |  |  if data[i] does not equal fresh then     /* We arrived at the next distinct value */
7  |  |  |  Append fresh to res ;            /* The previous fresh value is added to res  */
8  |  |  |  Assign to fresh the value of data[i];          /* This is the next fresh value */
9  |  |  end
10 |  |  Increment i by one;
11 |  end
12 |  Append fresh to res ;                    /* The last fresh value is appended to res  */
13 end
14 return res
```